Failure Detection for Wind Turbines

Our main goal for this project was to find failures in a wind turbine before it happened. While we did have access to the labels for the failure logs in event_info.csv, i tried using an unsupervised approach, by training on only healthy data, the VAE, was able to learn all the context of the data, more like “physics” of the turbine and was able to predict the failures by reconstructing the original input from the failure data.

Variational Auto Encoders

Main difference between Auto Encoders and Variational Auto Encoders in deep learning??
I like to use my favorite analogy here:
An Archer and his arch:

- For standard AE, every time he sees a specific wind speed and a power level, he shoots his target at exactly the dot (target).
- The problem is, even if the wind changes slightly, he just knows where the dot (target) is, because he memorizes the specific target. This causes him to hit in an empty area and then the model outputs garbage.
- For VAE, the archer aims for an area:

The Reparameterization Way:
We change the rules.

1. Archer (Encoder) chooses where to Aim (μ).
2. Archer (Encoder) chooses how "loose" his bowstring is (σ, stddev).
3. The Trick: We simply add a generic "Wind" (ϵ) that we calculate separately from a standard normal distribution (like rolling a dice on the side table).

The Formula:
$ArrowPosition(z) = Aim(\mu) + StringLooseness(\sigma) \times Wind(\epsilon)$

Result?
Because we have VAE that aims for an actual area, it has access to every single area, there are no empty spaces or gaps. The archer takes the wind and uses it to practice every single area. By hitting every shot in a specific area, this way circles will overlap, giving him the ability to learn every piece and context of that area, not like AE’s.

Code:

I first started with config.yaml,

```
paths:
 raw_data: "data"
 processed_data: "data/processed"
 logs: "logs/pipeline.log"
 outputs: "outputs/"

assets:
 target_asset: 50
 train_assets: [12,15,16]

feature_engineering:
 window_days: 14 # Tells the model to look at the 14 days leading up
to a failure to find patterns,
 # 1 hour(6) and 24 hours(144) for trends
 rolling_windows: [6, 144] #  6 Represents 1 hour of data (assuming
sensors record every 10 minutes). Represents 24 hours of data. This
captures slow wear and tear or temperature trends over a full day.

models:
 vae:
   latent_dim: 10 # This forces encoder to pick the 10 most important
"features" (like vibration frequency, heat trend, and pressure).
   hidden_dim: 256 # Needs to be larger to hold that info
   learning_rate: 0.0001
   epochs: 200
   batch_size: 64
   patience: 10 # stop if no improvement for 10 epochs
```

Here, data points to the csv file, pipeline.log refers to the log, and the output folder is where the graph visualizations were stored after training.
For our assets, for your target failure, I chose 50 assets, and for our training assets we just chose these, later we proved these are healthy.
For the featuring engineering section, i wanted to see trends like weather patterns that we have in our data, so i gave the 14 day block of data, that the model sees for every row of normal data.

Note: I later removed 144, rolling window cause of low ram resources, Only used 6 (1 hour window)

The Math Behind the Numbers

- For 1 Hour ("6"):
  - 60 minutes/10 minutes per sample=6 samples
  - This "Short Window" helps the VAE detect **sudden spikes** or "shocks" to the system (like a sudden jolt in vibration).
- For 24 Hours (The "144"):
  - 24 hours×6 samples per hour=144 samples
  - This "Long Window" helps the VAE see **gradual trends** (like a bearing slowly heating up over a day).

Remaining, for our model parameters,
For specifically the latent_dimension:

The Compression Way:

Why does this teach the AI "Physics"?
In a Wind Turbine:

- Input: Wind Speed (High), Power (High), Rotor Speed (Fast), Temperature (Hot)... (100 sensors).
- If we force the AI to compress this into 1 number, it can't memorize the temperature of every bolt. It has to find the Master Variable that causes everything else to happen.
- It realizes: "Wait, if Wind Speed is high, then Power is ALWAYS high. I don't need to store both numbers. I'll just store 'Energy Level' and I can reconstruct both later."

So, it combines to total 10 features, in simpler terms: This forces the encoder to pick the 10 most important "features" (like vibration frequency, heat trend, and pressure).

Then, for the hidden_dimension (the vae size), we took value of 256, gave it a learning rate of 0.0001, and then 200 epochs for training, and batch_size = 64 to process 64 one at a time, for processing smoothly, and patience of 10 counters for early stopping logic that i will take later.

src/data_loader.py:

This file contains the code to get the messy data csv's into proper python format.
We imported all libraries, then wrote the DataLoader class,
We had two functions that contributed to this file, one for loading the failure data, with debugging checks because I had trouble finding the asset_id columns, and later fixing it with the help of Ai assistance with Gemini.
The second function loads the turbine healthy data.

src/features.py:

The file essentially contains the transformation of features before they are ready for training.
We had a class FeatureEngineer, and in that, I implemented two key functions, one for creating the target feature code and the second one for the rolling features code.

The create_target function labels the data for training/validation.
1 for failure, and 0 for health.
Also i wrote code for grading the failures and normal, it looks back 14 days from a known failure and grades those rows.

The create_rolling_features function contains the code for transforming the raw sensor data to different contextual information features for each and every single sensor.
It takes approx. 100 raw sensor features and calculates the stddev and mean for over different time windows. It creates the roughly 4,755 features that give the vae enough contextual information about the turbines data, like for example, distinguishing between a normal day and a failing day.

Due to light resources and challenges, I made some changes for the math to use float32 instead of float64, to cut ram usage by 50%.

Next is src/utils_deep.py:

Since this is a deep learning project, these models only understand through tensors and not dataframes.

Creating this because the raw data is currently in csv format and I want to convert to tensors.
Matrix: 2d grid
Tensor: N-dimensional grid, 3d, 4d, etc.

Basic things, imported libraries, and then made a class.

It removes any sensor columns that never change, like 0 variance. We dropped these to prevent NaN errors.
We used standard scaling, Prevents big numbers (Power levels) from dominating small numbers (Vibration levels). This is critical because it ensures a small vibration signal isn't ignored just because a power sensor is huge, like 1000.

StandardScaler does this to sensor data:

- Subtracts the Mean: This slides the whole graph left or right so it's centered at zero.
- Divided by Standard Deviation: This shrinks or stretches the graph so it isn't too tall or too short.

Then splitting data into **Training** (80%) and **Validation** (20%) to test its progress.

Next is src/model_vae.py:

This is basically the architecture of our VAE.

Firstly, there is an encoder: It takes the high dimensional input then shrinks it down through the hidden layer: 256 nodes to find mean and log_variance. Note: I have used log_variance instead of stddev because the output of stdfdev may be negative so i used log_variance, when exponentialted, results in a positive number.

The reparametrization trick, i used:
I used the formula previously z = mu + sigma * epsilon
It allows the model to practise using random noise (epsilon), this is the wind we talked about previously in our archer analogy.

Finally, the decoder after finishing this process takes the 10 compressed features and tries to reconstruct back to the original 2,000 features. Note: I have mentioned 2,000 features and note 4,000 because I had to remove the 144 (24-hour rolling window), due to low vram issues and challenges we faced.

Next is src/trainer_vae.py:

Trainer

We use two approaches: Reconstruction Loss and KL Divergence

1) **Reconstruction Loss**
   This part measures how well the model can reconstruct the original input.
   Here, our objective is to minimize the difference between the original input and the reconstructed output.
   We are using MSE for this.
2) **KL Divergence**
   KL Divergence Loss forces the model to organise its findings into a specific shape–usualy standard normal deviation (like a bell curve).
   Here, our objective is to make the space predictable and continuous.

By forcing everything into a bell curve:

- **No Isolation:** Every point is surrounded by other valid points.
- **Interpolation:** The model is forced to make the transition from one point to another smooth.

KLD Formula

$$KLD = -1/2\,(1 + ln(sigma\hat{}2) - mu\hat{}2 - sigma\hat{}2$$

It is the closed-form solution for the Kullback-Leibler Divergence between two Gaussian Distributions:

1. Our predicted distribution
2. N(μ,σ^2)
   For every image I feed to the model, the encoder says: I think this image belongs to this average location (mu), with this much uncertainty (sigma^2).
3. The Standard Normal distribution
4. N(0,1)
   Our target we want, where the bell curve is perfectly at the center (0) with width of 1.

We are mathematically measuring the "distance" between our messy predicted cloud and a perfect, clean sphere. We minimize this distance to keep the latent space organized.

Note: I found this useful, for a possible challenge: My VAE is reconstructing the images perfectly, but when I generate new samples, they look like garbage. What is wrong?
Answer: because the kl divergence weight is too low. The model prioritized mse more than kld.

Also, i applied early stopping logic here:
Epoch: One pass through the entire dataset
Overfitting: When the model starts memorizing the noise of the training data instead of learning physics.

Early stopping: We hold back a small piece of data, we watch the error in this validation set. If the training error goes down but validation error goes up, the model is overfitting. We stop immediately.
Additional Notes:
Early stopping logic in src/trainer_vae.py
In this following block
```
best_val_loss = float('inf')
        patience_counter = 0
        best_model_state = None
```

In the above block, best val loss's value is representing a value 'infinity'
When the first epoch finishes, any real number for loss (like 500.0) will be smaller than infinity. This ensures that the model records its very first attempt as the "current best" and then tries to lower that number in every future epoch.
If the epoch gets better and the avg_val_loss does not get better than the best_val_loss, we give it a strike counter of 1. If the strike counter reaches the limit we gave in config.yaml, the training stops.

For the best_model_state variable,
The Logic: Every time the model hits a new "Best Score," we use copy.deepcopy(self.model.state_dict()) to take a snapshot of its brain at that exact moment and save it in this variable.
The Result: Even if the model eventually fails or over-trains, we can "time travel" back to the best version of the model before we finish the function.

Early Stopping Logic code:
```
if avg_val_loss < best_val_loss:
               best_val_loss = avg_val_loss
               patience_counter = 0
               # Save the best model in memory
               best_model_state =
copy.deepcopy(self.model.state_dict())
           else:
               patience_counter += 1
               if patience_counter >= self.patience:
                   self.logger.info(f"Early Stopping triggered at
Epoch {epoch}!")
                   Break
```

Here, pretty self-explanatory, if the average loss is improving and good than the previous best validation loss, save the best val loss to the lowest average loss, reset the patience counter every time that occurs, and save the model to the best model state, else, add 1 everytime it does not improve, and break if reached the final patience state.

In main.py,

This code:

```
 # Create features (Rolling Mean/Std)

                df = engineer.create_rolling_features(df)
```

A single sensor reading like 80 degree celsius doesn't tell you if it is stable or failing, by using our rolling features from the features.py we created, it calculates the rolling mean and rolling standard deviation of the single sensor reading. (6 and 144 steps)

Result: It transforms a simple column like vibration into three columns: vibration, vibration_mean, and vibration_std. This gives the VAE "context" so it can see trends and shaking patterns over time rather than just a single snapshot.

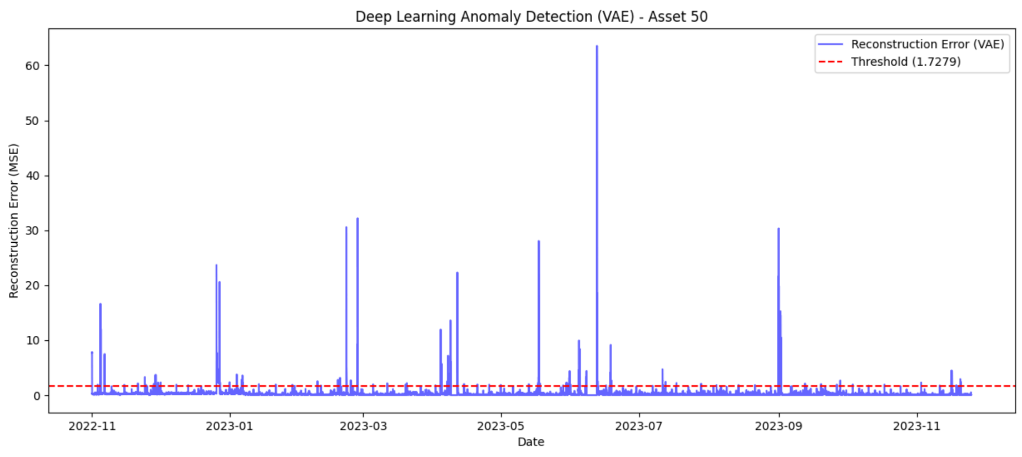
Challenges

My biggest challenge was gpu and memory so i did this:
I hit a memory bottleneck where feature expansion exceeded 30GB RAM. Instead of throwing more compute at it, I analyzed the feature importance and realized the 24-hour rolling window was redundant for high-frequency anomaly detection. I optimized the pipeline to prune those features at runtime, reducing memory footprint by 50% without sacrificing recall.

Because of that cut, I was able to:

1. **Finish Training:** It stopped crashing and successfully ran for those 4-5 hours.
2. **Maintain Accuracy**
3. **Prove Efficiency**

Report Visualization:


Deep Learning Anomaly Detection (VAE) - Asset 50
Reconstruction Error (VAE)
Threshold (1.7279)
Reconstruction Error (MSE)
Date